

---

# High-Performance Distributed Machine Learning using Apache SPARK

---

**Celestine Dünner**

IBM Research – Zürich, Switzerland

CDU@ZURICH.IBM.COM

**Thomas Parnell**

IBM Research – Zürich, Switzerland

TPA@ZURICH.IBM.COM

**Kubilay Atasu**

IBM Research – Zürich, Switzerland

KAT@ZURICH.IBM.COM

**Manolis Sifalakis**

IBM Research – Zürich, Switzerland

EMM@ZURICH.IBM.COM

**Haralampos Pozidis**

IBM Research – Zürich, Switzerland

HAP@ZURICH.IBM.COM

## Abstract

In this paper we compare the performance of distributed learning using Apache SPARK and MPI by implementing a distributed linear learning algorithm from scratch on the two programming frameworks. We then explore the performance gap and show how SPARK-based learning can be accelerated, by reducing computational cost as well as communication-related overheads, to reduce the relative loss in performance versus MPI from  $20\times$  to  $2\times$ . With these different implementations at hand, we will illustrate how the optimal parameters of the algorithm depend strongly on the characteristics of the framework on which it is executed. We will show that carefully tuning a distributed algorithm to trade-off communication and computation can improve performance by orders of magnitude. Hence, understanding system aspects of the framework and their implications, and then correctly adapting the algorithm proves to be the key to performance.

## 1. Introduction

Machine learning techniques provide consumers, researchers and businesses with valuable insight. However, significant challenges arise when scaling the corresponding training algorithms to massive datasets that do not fit in the memory of a single machine, making the implementation as well as the design of distributed algorithms more demanding than that of single machine solvers. Going distributed involves a lot of programming effort and system-

level knowledge to correctly handle communication and synchronization between single workers, and, in addition, requires carefully designed algorithms that run efficiently in a distributed environment. In the past decades distributed programming frameworks such as Open MPI have empowered rich primitives and abstractions to leverage flexibility in implementing algorithms across distributed computing resources, often delivering high-performance but coming with the cost of high implementation complexity. In contrast, more *modern* frameworks such as Hadoop and SPARK have recently emerged which dictate well-defined distributed programming paradigms and offer a powerful set of APIs specially built for distributed processing. These abstractions make the implementation of distributed algorithms more easily accessible to developers, but seem to come with poorly understood overheads associated with communication and data management which make the tight control of computation vs communication cost more difficult. In this work we will analyze these overheads and show that to minimize their effect – and thus to design and implement efficient real-world distributed learning algorithms using Apache SPARK – it is important to be aware of these overheads and adapt the algorithm to the conditions given. Hence, understanding the underlying system and adapting the algorithm, remains the key to performance even when using Apache SPARK and we demonstrate that a careful tuning of the algorithm parameters can decide upon several orders of magnitude in performance. The three main contributions of this paper are the following:

- We provide a fair analysis and measurements of the overheads inherent in SPARK, relative to an equivalent MPI implementation of the same linear learning

algorithm. In contrast to earlier work (Reyes-Ortiz et al., 2015; Gittens et al., 2016; Ousterhout et al., 2015) we clearly decouple framework-related overheads from the computational time. We achieve this by off-loading the critical computations of the SPARK-based learning algorithm into compiled C++ modules. Since the MPI implementation uses exactly the same code, any difference in performance can be solely attributed to the overheads related to the SPARK framework.

- We demonstrate that by using such C++ modules we can accelerate SPARK-based learning by an order of magnitude and hence reduce its relative loss in performance over MPI from  $20\times$  to an acceptable level of  $2\times$ . We achieve this by, firstly, accelerating computationally heavy parts of the algorithm by calling optimized C++ modules from within SPARK, and, secondly, utilizing such C++ modules to extend the functionality of SPARK to reduce overheads for machine learning tasks.
- Our clear separation of communication and computation related costs provides new insights into how the communication-computation trade-off on real world systems impacts the performance of distributed learning. We will illustrate that if the algorithm parameters are not chosen carefully – in order to trade-off computation versus communication – then this can lead to performance degradation of over an order of magnitude. Furthermore, we will show that the optimal choice of parameters depends strongly on the programming framework the algorithm is executed on: the optimal choice of parameters differs significantly between the MPI and the SPARK implementations respectively, even when running on the same hardware.

For this analysis we will focus on standard distributed learning algorithms using synchronous communication, such as distributed variants of single machine solvers based on the principle of mini-batches, e.g., mini-batch stochastic gradient descent (SGD) (Dekel et al., 2012; Zinkevich et al., 2010) and mini-batch stochastic coordinate descent (SCD) (Richtarik & Takac, 2015; Shalev-Shwartz & Zhang, 2013), as well as the recent CoCoA method (Jaggi et al., 2014; Ma et al., 2015; Smith et al., 2016). In contrast to more complex asynchronous frameworks such as parameter servers (Li et al., 2014; Microsoft, 2015), synchronous schemes have tighter theoretical convergence guarantees and are easier to accurately benchmark, allowing for a more isolated study of system performance measures, such as the communication bottleneck and framework overheads. It is well known that for such methods, the mini-batch size serves as a tuning parameter to efficiently control the trade-off between communication and computation – where we

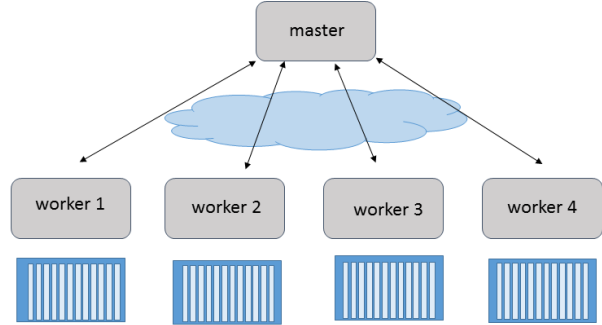


Figure 1. Data partitioned across multiple machines in a distributed environment. Arrows indicate the synchronous communication per round, that is, sending one vector update from each worker, and receiving back the resulting sum of the updates (AllReduce communication pattern in MPI and SPARK).

will see that the optimal mini-batch size shows a very sensitive dependence on the algorithm and the programming framework being used. For our experiments we have chosen the state of the art CoCoA algorithm and implemented the same algorithm on the different programming frameworks considered. CoCoA is applicable to a wide range of generalized linear machine learning problems. While having the same flexible communication patterns as mini-batch SGD and SCD, CoCoA improves training speed by allowing immediate local updates per worker, leading to up to  $50\times$  faster training than standard distributed solvers such as those provided by MLlib (Meng et al., 2016). Our results can provide guidance to developers and data scientists regarding the best way to implement machine learning algorithms in SPARK, as well as provide valuable insight into how the optimal parameters of such algorithms depend on the underlying system and implementation.

## 2. The Challenge of Distributed Learning

In distributed learning, we assume that the data is partitioned across multiple worker nodes in a cluster, and these machines are connected over a network to the master node as illustrated in Figure 1. We wish to learn the best classification or regression model from the given training data, where every machine has access to only part of the data and some shared information that is periodically exchanged over the network. This periodic exchange of information is what makes machine learning problems challenging in a distributed setting because worker nodes operate only on a subset of the data, and unless local information from every worker is diffused to every other worker the accuracy of the final model can be compromised. This exchange of information however is usually very expensive relative to computation. This challenge has driven a significant effort in recent years to develop novel methods en-

abling communication-efficient distributed training of machine learning models.

Distributed variants of single machine algorithms such as mini-batch SGD and SCD are well known work-horses in this context, where, in every round, each worker processes a small fixed number  $H$  of local data samples, in order to compute an update to the parameter vector, see e.g. (Dekel et al., 2012; Zinkevich et al., 2010) and (Richtarik & Takac, 2015; Shalev-Shwartz & Zhang, 2013). The update of each worker is then communicated to the master node, which aggregates them, computes the new parameter vector and broadcasts this information back to the workers. A useful property of the mentioned algorithms is that the size of the mini-batch,  $H$ , can be chosen freely. Hence,  $H$  allows control of the amount of work that is done locally between two consecutive rounds of communication. On a real-world compute system, the costs associated with communication and computation are typically very different and thus the parameter  $H$  allows to optimally trade-off these costs to achieve the best overall performance.

## 2.1. Algorithmic Framework

Similar to mini-batch SGD and SCD, the state of the art CoCoA framework (Jaggi et al., 2014; Ma et al., 2015; Smith et al., 2016) allows the user to freely trade-off communication versus computation. For our study, we chose the CoCoA algorithm for training of generalized linear machine learning models, because of its superior performance over mini-batch methods, and because it still offers a similar tuning parameter  $H$ , for the number of points processed per communication round, which allows a fair investigation of the communication-computation trade-off and the associated overheads. The CoCoA algorithm is designed to efficiently solve regularized linear loss minimization problems of the form

$$\min_{\alpha \in \mathbb{R}^n} \ell(A\alpha) + r(\alpha) \quad (1)$$

in a distributed setting. Here  $\alpha$  parametrizes the model,  $\ell$  and  $r$  are convex functions and  $A$  is the matrix where the number of rows equals the number of datapoints,  $m$ , and the number of columns equals the number of features,  $n$ . During training with CoCoA, every worker node repeatedly works on a local approximation of (1) based on its locally available data, after which it communicates a single vector of updates back to the master. Every message sent will be a vector of dimensionality  $m$ . To solve the local subproblem, CoCoA allows the user to employ an arbitrary local solver on each worker. The amount of time spent computing in the local solver between two consecutive rounds of communication can then be controlled by adapting the parameter,  $H$ , which determines the number of local points processed in each round, and thus steering

the accuracy to which each local subproblem is solved. For more detail about the algorithm framework and subproblem formulations, we refer to (Smith et al., 2016). In this paper we will choose SCD as a local solver, where every node works on its dedicated coordinates of  $\alpha$ . In this case CoCoA differs from classical mini-batch SCD (a.k.a. SDCA, see (Shalev-Shwartz & Zhang, 2013; Richtarik & Takac, 2015)) in that coordinate-updates are immediately applied locally.

## 3. Programming Frameworks for Distributed Computing

There exist many different programming frameworks that are designed to simplify the implementation of distributed algorithms. In this work we will focus on SPARK, due to its widespread use, and compare it against the well established MPI framework.

### 3.1. SPARK

Apache SPARK (Zaharia et al., 2012) is an open source general-purpose cluster computing framework developed by the AMP lab at the University of California, Berkeley. The core concept underpinning SPARK is the resilient distributed dataset (RDD). An RDD represents a read-only collection of elements, spread across a cluster that can be operated on in parallel. Through the concept of the lineage graph, RDDs provide fault tolerance by keeping transformation logs. With this abstraction, SPARK allows the developer to describe their distributed program as a sequence of high level operations on RDDs without being concerned with scheduling, load-balancing and fault tolerance. These high-level operations are of two types: transformations and actions. Whereas transformations create a new RDD from one or more existing RDDs, actions launch a computation on an RDD and generate an output. Transformations are evaluated *lazily*, this means that SPARK waits to execute transformations until an action is called to generate an output. This allows the SPARK engine to do simple optimization and pipelining within stages.

**pySPARK.** The core of SPARK is written in Scala, runs on the Java\* virtual machine (JVM) and offers a functional programming API to Scala, Python, Java and R. The python API is called pySPARK and exposes the SPARK programming model to Python. Specifically, the local driver consists of a Python program in which a SPARK context is created. The SPARK context communicates with the Java virtual machine (over py4J) which in turn is responsible for initiating and communicating with Python processes.

**SPARK MLlib.** MLlib is the machine learning library built on top of SPARK, designed to harness its scalability

and performance properties. It implements commonly used algorithms for machine learning including regression, classification, clustering, dimensionality reduction and collaborative filtering.

### 3.2. MPI

Message Passing Interface (MPI) (Forum, 1994) is a language-independent communications protocol for parallel computing that has been developed for high-performance computing (HPC) platforms. It offers a scalable and effective way of programming distributed systems consisting of tens of thousands of nodes. The collective communication functions of MPI (e.g., broadcast, gather, scatter, and reduce operations) significantly simplify the design of distributed machine learning algorithms. MPI allows application programmers to take advantage of problem-specific load-balancing and communication optimization techniques. MPI is independent of the underlying network stack. It can operate on top of TCP/IP, RDMA, or on top of high-performance network architectures supported by supercomputers. Furthermore, MPI offers various different ways of enabling fault-tolerance for distributed applications. However, enabling fault-tolerance, customizing load-balancing, and minimizing the communication overhead typically requires a significant amount of manual work and necessitates advanced understanding of the algorithms, MPI's library functions, and the underlying network architecture.

## 4. Implementation Details

To compare and analyze the performance of the aforementioned programming frameworks, we have implemented the CoCoA algorithm (Smith et al., 2015) from scratch on SPARK, pySPARK and MPI. In our implementations these program frameworks are used to handle the communication of updates between workers during the training of CoCoA. Additionally, SPARK handles data partitioning and data management, which needs to be implemented by the user in the case of MPI.

**Data Partitioning.** In our implementations we distribute the data matrix  $A \in \mathbb{R}^{m \times n}$  column-wise according to the partition  $\{\mathcal{P}_k\}_{k=1}^K$ . Hence, columns  $\{\mathbf{c}_i\}_{i \in \mathcal{P}_k}$  reside on worker  $k$ , where  $n_k = |\mathcal{P}_k|$  denotes the size of the partition. Thus, every worker samples uniformly at random from its  $n_k$  local features (also referred to as  $n_{local}$ ) and updates the corresponding coordinates  $\alpha_{[k]}$  of the parameter vector. We use the notation  $\alpha_{[k]}$  to denote the vector  $\alpha$  with nonzero entries only at positions  $i \in \mathcal{P}_k$ .

**Communication Pattern.** During the CoCoA algorithm execution, after every computation round consisting of  $H$

local coordinate steps, the individual workers communicate a single  $m$ -dimensional vector

$$\Delta \mathbf{v}_k := A \Delta \alpha_{[k]},$$

to the master node, where  $\Delta \alpha_{[k]}$  denotes the update computed by worker  $k$  to its local coordinate vector during the previous  $H$  coordinate steps.  $\Delta \mathbf{v} \in \mathbb{R}^m$  is a dense vector, encoding the information about the current state of  $\alpha_{[k]}$ , where  $\alpha_{[k]}$  itself can be kept local. The master node then aggregates these updates and determines  $\mathbf{v}^{(t+1)} = \mathbf{v}^{(t)} + \sum_k \Delta \mathbf{v}_k$  which is then broadcast to all workers to synchronize their work. The definition of the shared vector  $\mathbf{v} := A\alpha$  is motivated by Fenchel-Rockafellar duality, where  $\mathbf{v}$  is related to the dual parameter vector which is defined through the linear map  $A\alpha$ , see (Bauschke & Combettes, 2011; Dünner et al., 2016) for more details.

### 4.1. Implementations

In this section we present five selected implementations of CoCoA for the widely used problem of ridge regression – that is  $\ell$  being the linear least-squares cost function, and  $r(\cdot) := \lambda \|\cdot\|^2$  being the Euclidean norm regularizer – under the assumptions described in the previous section. Mathematically, all our algorithm implementations are equivalent, but small differences in the learned model can occur due to different random number generation, floating point precision and slight variations in the partitioning of the data across workers. The first four implementations are all based on SPARK but make use of different language bindings and in some cases leverage APIs to expose data stored within RDDs to native code. The fifth implementation is built on MPI as a reference.

**A) Spark.** We use the open source implementation of Smith et al. (Smith & Jaggi, 2015) as a reference implementation of CoCoA. This implementation is based on SPARK and entirely written in Scala. The Breeze library (Hall et al.) is used to accelerate sparse linear algebra computations. As SPARK does not allow for persistent local variables on the workers, the parameter vector  $\alpha$  needs to be communicated to the master and back to the worker in every round, in addition to the shared vector  $\mathbf{v}$  – the same applies to implementations (B), (C) and (D).

**B) Spark+C.** We replace the local solver of implementation (A) with a Java native interface (JNI) call to a compiled and optimized C++ module. Furthermore, the RDD data structure is modified so that each partition consists of a flattened representation of the local data. This modification allows one to execute the local solver using a *map* operation in Spark instead of a *mapPartitions* operation. In that manner, one can pass the local data into the native function call as pointers to contiguous memory regions rather than

having to pass an iterator over a more complex data structure. The C++ code is able to directly operate on the RDD data (with no copy) by making use of the *GetPrimitiveArrayCritical* functions provided by the JNI.

**C) pySpark.** This implementation is equivalent to that of (A) except it is written entirely in Python/pySPARK. The local solver makes use of the *NumPy* package (Walt et al., 2011) for fast linear algebra.

**D) pySpark+C.** We replace the local solver of implementation (C) with a function call to a compiled and optimized C++ module, using the Python-C API. Unlike implementation (B) we did not flatten the RDD data structure since this was found to lead to worse performance in this case. Instead, the local solver is executed using a *mapPartitions* operation. Within the *mapPartitions* operation we iterate over the RDD in order to extract from each record a list of *NumPy* arrays. Each entry in the list contains the local data corresponding to a given feature. The list of *NumPy* arrays is then passed into the C++ module. The Python-C API allows *NumPy* arrays to expose a pointer to their raw data and thus the need to copy data into any additional C++ data structures is eliminated.

**E) MPI.** The MPI implementation is entirely written in C++. To initially partition the data we have developed a custom load-balancing algorithm to distribute the computational load evenly across workers, such that  $\sum_{i \in P_k} \text{#nonzeros}(\mathbf{c}_i)$  is roughly equal for each partition. Such a partitioning ensures that each worker performs roughly an equal amount of work and was found to perform comparable to the SPARK partitioning.

Note that the C++ code that implements the local solver in implementations (B), (D) and (E) is identical up to specific JNI/Python-C API functions.

## 4.2. Infrastructure

For the experiments discussed in the next section we ran our algorithm implementations on a cluster of 4 physical nodes interconnected in a LAN topology through a 10Gbit-per-port switched inter-connection. Each node is equipped with 64GB DDR4 memory, an 8-core Intel Xeon® E5 x86\_64 2.4Ghz CPU and solid-state disks using PCIe NVMe 3.0 x4 I/O technology. The software configuration of the cluster is based on Linux® kernel v3.19, MPI v3.2, and Apache Spark v1.5. Spark is configured not to use the HDFS filesystem; instead SMB sharing directly over ext4 filesystem I/O is employed. While this decision may occasionally give reduced performance in Spark, on one hand it eliminates I/O measurement delay-variation artifacts due to the extensive buffering/delay-writing of streams in HDFS, and on the other hand it enables more *fair* comparison

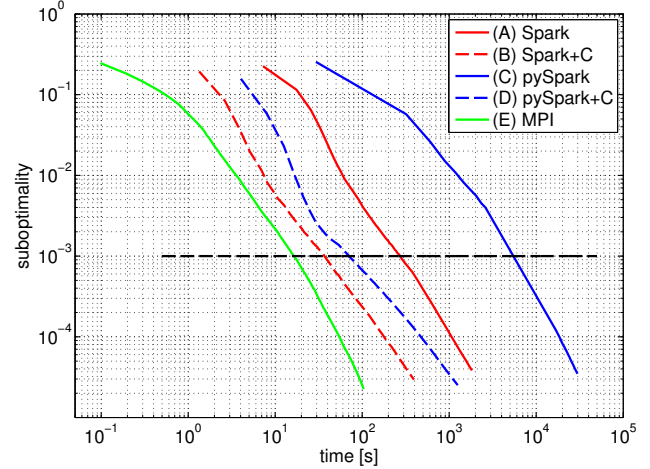


Figure 2. Suboptimality over time of implementations (A)-(E) for training the Ridge Regression model on webspam.

with MPI since all overheads measured are strictly related to Spark. Finally, all cluster nodes are configured without a graphical environment or any other related services that could possibly compete with Spark or MPI over CPU, memory, network, or disk resources.

## 5. Experimental Results

We investigate the performance of the five different implementations of the CoCoA algorithm discussed in Section 4, by training a ridge regression model on the publicly available *webspam* dataset<sup>1</sup>. All our experiments are run on our internal cluster described in Section 4.2. If not specified otherwise, we use 8 SPARK workers with 24 GB of memory each, 2 on each machine, which allows the data partitions to fit into memory. All our results are shown for optimized parameters, including  $H$ , to suboptimality  $\epsilon = 10^{-3}$  and the results are averaged over 10 runs.

### 5.1. SPARK Performance Study

Figure 2 gives an overview over the performance of implementation (A)-(E), showing how the suboptimality evolves over time during training for every implementation. We see that the reference SPARK code, (A), written in Scala performs significantly better than the equivalent Python implementation, (C). This is to be expected, for two main reasons: 1) Scala is a JVM compiled language in contrast to Python, 2) SPARK itself is written in Scala and using pySPARK, adds an additional layer which involves data copy and serialization operations.

In this paper we would like to study the overheads present in the SPARK framework in a language independent manner (in as far as it is possible). As described in Section 4.2,

<sup>1</sup><http://www.cc.gatech.edu/projects/doi/WebbSpamCorpus.html>



this can be achieved by offloading the computationally intense local solvers into compiled C++ modules for both the Scala as well as the Python implementations. In Figure 2 the performance of these new implementations is shown by the dashed lines. As expected, the performance gain is larger for the Python implementation. However, the Scala implementation can also significantly benefit from these extensions: the performance gap between MPI and SPARK is reduced from  $10\times$  to  $4\times$ .

## 5.2. SPARK overheads

To accurately measure the framework overhead of SPARK and pySPARK, we leverage the fact that the accelerated SPARK, pySPARK as well as the MPI implementation – (B), (D) resp. (E) – execute exactly the same C++ script on the workers, in each round. Hence, the difference in performance of the red and blue dashed lines in Figure 2 can – ignoring the minor computational work on the master – entirely be attributed to the overheads involved in using the python API. Similarly, the difference in performance of the dashed lines to the MPI implementation can be traced to the overheads of the SPARK and pySPARK frameworks, respectively, over MPI.

To get a better understanding of these overheads and separate them from the computation time, we ran every implementation for 100 rounds using  $H = n_{\text{local}}$  and measured the execution time for the following three sections of the code:

$T_{\text{tot}}$	:	Total run time
$T_{\text{worker}}$	:	Time spent computing on the workers
$T_{\text{master}}$	:	Time spent computing on the master
$T_{\text{overhead}}$	:	Overheads $:= T_{\text{tot}} - T_{\text{worker}} - T_{\text{master}}$

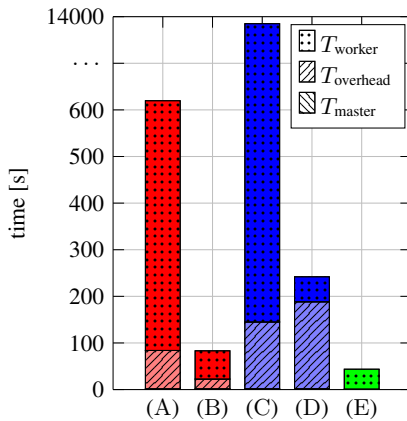


Figure 3. Extract execution times and overheads for 100 iterations with  $H = n_{\text{local}}$  for the SPARK implementations (A) and (B), the pySPARK implementations (C) and (D) and the MPI implementation (E).

The results of this analysis are displayed in the bar plot in Figure 3. The first observation we make is that the time spent computing on the master is very small ( $< 2s$ ) for all implementations and the execution time splits into worker computation time and overheads related to communications and data serialization/de-serialization.

Firstly, focusing on the worker execution times, shown by the dotted areas, we note, that the performance of the SPARK and pySPARK implementations, (A) and (C), is vastly dominated by the time spent in the local solver. The benefit from replacing the local solver by C++ modules, i.e.  $(A, C) \rightarrow (B, D)$ , is significant and reduces the local execution time of the SPARK implementation by one order of magnitude and the execution time of the pySPARK implementation by more than 2 orders of magnitude. The local execution time of the C++ code is roughly the same for implementation (B), (D) and (E). When going into more detail, however, we can see that despite executing identical code, we observe a slight increase in the worker execution time for implementation (B). While the source of this is not known we suspect it due to the internal workings of the JNI.

Now focusing on the framework overheads, shown by the dashed areas in Figure 3, we can see that the overheads of the pySPARK implementation are  $15\times$  larger than those of the reference SPARK implementation written in Scala. This performance loss of pySPARK was also observed in earlier work, i.e. (Karau, 2016), and comes from the Python API which adds additional serialization steps and overheads coming with initializing Python processes and copying data from the JVM to the Python interpreter. Newer versions of SPARK 1.5.2 offer dataframes, which promise to alleviate some of overheads of pySPARK (Armbrust et al., 2015), but they are not yet ready to be used in machine learning applications, due to lack of fast iteration operators on top of dataframes. Further, we see that calling the C++ modules from Python adds some additional overhead on top of the pySPARK overhead, coming from minimal data copy operations into C++ memory as well as multiple calls to the Python-C API. However, these overheads are negligible compared to the gain in execution time achieved by using these extensions. For Scala, we do not have this direct comparison as we changed the data format of the reference implementation when adding the C++ modules, in order to minimize the number of JNI calls. We see that this flat data format is much more efficient for the Scala implementation and reduces overheads by a factor of 3. We have also implemented this format in Python but we could not achieve a similar improvement. For MPI the overheads are negligible and only account for 3% of the total execution time.

### 5.3. Reducing SPARK Overheads

As we have seen, SPARK, especially pySPARK, adds significant overhead to the execution of the learning algorithm as opposed to MPI. In this section we will propose two techniques for extending the functionality of SPARK so that these overheads can be somewhat alleviated.

**Addition of Persistent Local Memory.** SPARK does not allow for local variables on the workers that can persist across stage boundaries, that is the algorithm rounds. Thus, the CoCoA algorithm has to be implemented in SPARK involving additional communication, since it is not possible for workers to store their dedicated coordinates of  $\alpha$  locally. As a consequence, in addition to the shared vector, the  $\alpha$  vectors need to be communicated to the master and back in every stage of the algorithm, thus increasing the overhead associated with communication.

However, one can additionally implement such functionality from within the C++ extension modules. Globally-scoped C++ arrays can be allocated upon first execution of the local solver that store the local  $\alpha$  vectors. The state of these arrays persists into the next stage of execution in SPARK, and thus the additional communication is no longer necessary – this of course comes at a small expense of a violation of the SPARK programming model in terms of consistency of external memory with the lineage graph.

**Meta-RDDs** For the Python implementations in particular, there is a significant overhead related to the RDD data structure and its elements themselves. It is possible to overcome this overhead by following the approach of (Hunter, 2016) and working with RDDs that consist purely of metadata (e.g. feature indices) and handling all loading and storage of the training data from within underlying native functions. While this may now be considered a serious deviation from the SPARK programming model (data resiliency is most likely lost), SPARK is still being used to schedule execution of the local workers, to collect and aggregate the local updates and broadcast the updated vectors back to the workers.

We have implemented both of these features for both the Scala and the Python-based implementations. In Figure 4 we compare the execution time and the overheads of these optimized implementations (B)\* and (D)\* with the corresponding implementations that make use of native functions but do not break with the SPARK programming model. We observe that our two modifications reduce overheads of the Scala implementation, (B), by 3 $\times$  and those of the Python implementation, (D), by 10 $\times$ . For the Scala implementation, the overall improvement due to using the meta-RDD is negligible and most of the gain comes from communication of less data. However, for the Python imple-

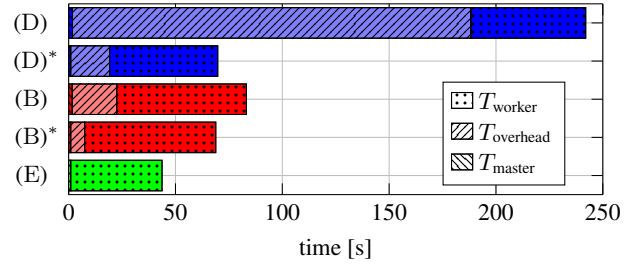


Figure 4. Overhead and compute time for the MPI implementation (E), the SPARK implementation (B), the pySPARK implementation (D) and our optimized implementations (B)\* and (D)\*.

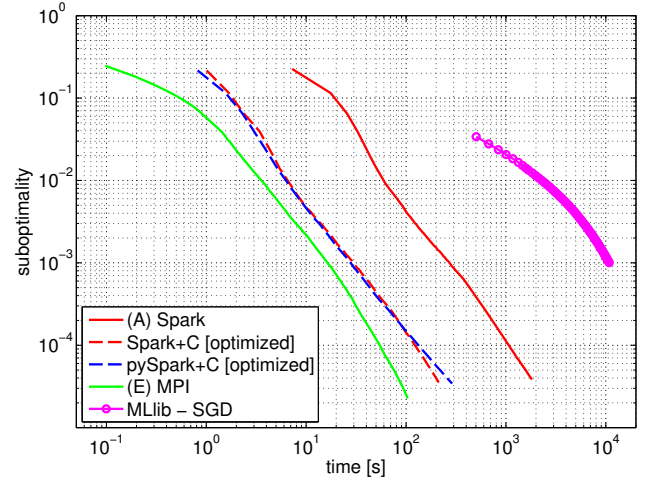


Figure 5. Performance gain of the optimized SPARK implementation over the reference implementation (A) and the MLLib solver.

mentation the effect of using meta-RDDs is far more significant. This is most likely due to the vast reduction in intra-process communication that has been achieved. In Figure 5 we illustrate that with our optimizations we achieve a 10 $\times$  speedup over the reference implementation and the performance is less than a factor of 2 slower than MPI. We also see that by implementing such extensions the performance of SPARK and pySPARK is more or less equivalent.

Note that the performance gain from reducing overheads in distributed algorithms not only comes from less time spent in every communication round but also from tuning  $H$  to the reduced communication cost. This enables more frequent communication, i.e. decrease  $H$ , and benefit from better convergence in the CoCoA algorithm. The effect of tuning  $H$  will be studied in detail in Section 5.5.

### 5.4. MLLib Solvers

In Figure 5 we compare the performance of our implementation against the performance of the MLLib solver available in pySPARK. We used the Linear Regression solver

from the library which is based on SGD and we tuned its batch size to get the best performance. Our findings are consistent with earlier results (Smith et al., 2015) that show that the reference CoCoA implementation outperforms existing state-of-the-art distributed solvers by up to  $50\times$ . With our optimizations, we gain another order of magnitude over the MLlib solver which enables training of the ridge regression model on our sample dataset in the order of minutes as opposed to hours.

### 5.5. Communication-Computation Trade-Off

We have seen in Figure 3 that the different implementations (A)-(E) suffer from different overheads associated with communication and data management. To fairly compare the performance of the different implementations in Figure 2 and Figure 5 we have optimized  $H$  separately for each implementation to account for the different communication and computation costs. In this section we study in more detail how  $H$  – the number of points processed locally in every round – can be used to control the trade-off between communication and computation and how this impacts the performance of CoCoA.

Figure 6 shows the time needed to achieve a suboptimality of  $10^{-3}$  as a function of  $H$  for the five different implementations (A)-(E). We see that for the different implementations the optimal value of  $H$  is indeed different. Hence, in order to get the best performance out of every implementation,  $H$  needs to be tuned individually to the respective communication and computation costs, whereas not doing so may degrade performance dramatically. We can see that, in our setting, the best performance of the pySPARK implementation is achieved for  $H = 0.2n_{\text{local}}$ , i.e., every worker performs  $0.2n_{\text{local}}$  coordinate updates in every round. For the accelerated pySPARK implementation, (D), however, the optimal value of  $H$  is more than  $25\times$  larger. This is

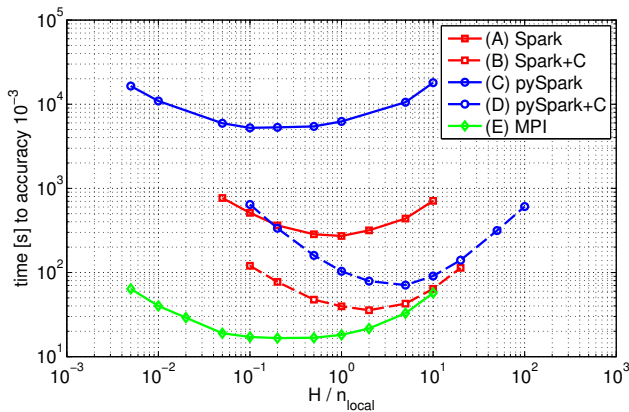


Figure 6. Time to achieve training suboptimality  $10^{-3}$  for implementations (A)-(E) as a function of  $H$ .

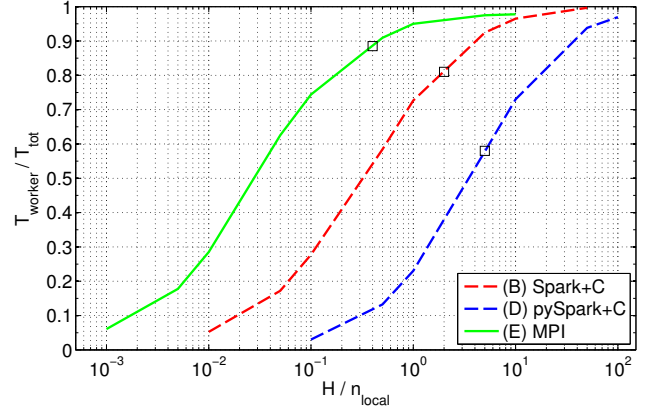


Figure 7. Fraction of time spent computing vs.  $H$  for implementations (B), (D) and (E).

because in implementation (D) the computational cost is significantly reduced as compared to the vanilla pySPARK implementation, see Figure 3, and we can afford to do more updates between two consecutive rounds of communication to find a more accurate solution to the local subproblems. Comparing now implementation (D) to the MPI implementation, (E), where communication is much cheaper, see Figure 3, we can conclude that communicating more frequently to benefit from improved convergence leads to better performance. For the Scala implementation, shown by the red curves, the same reasoning applies, whereas overheads are less significant.

These results show that trading-off communication and computation is crucial in designing and applying distributed algorithms. For example, as the optimized  $H$  value for implementation (E) is not optimal for implementation (D), it would more than double its training time.

Figure 7 shows how  $H$  impacts the trade-off between communication and computation for implementations (B), (D) and (E) by illustrating the fraction of time spent within the local solver as a function of  $H$ . The open squares indicate the optimal  $H$  found in Figure 6 for the corresponding implementations. We again see that for the SPARK implementations, which suffer from higher communication overheads, larger  $H$  are needed to strike a good balance between communication and computation. However, as  $H$  increases the benefit of doing additional updates between communication rounds and further improve the accuracy to which the local subproblem is solved, vanishes. Therefore, we can see that the optimal ratio between computation time and communication overhead is different for every implementation. Whereas for MPI we ideally spend up to 90% of the time computing, for the pySPARK+C implementation 60% appears to be optimal. This optimal fraction decreases with increasing effective overheads – which makes the algorithm suffer increasingly from the communication bottleneck.



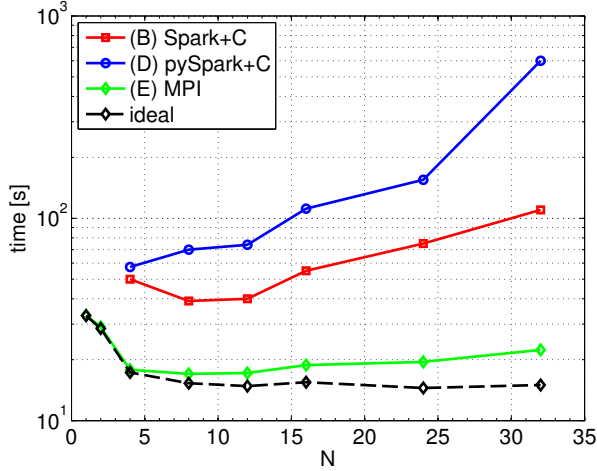


Figure 8. Time to reach an accuracy  $10^{-3}$  as a function on the number of worker nodes,  $N$ .

### 5.6. Scaling Behavior

In this subsection we investigate how the overheads of the different implementations impact the scaling behavior of distributed learning algorithms. To that end, in Figure 8, we plot the time the COCoA algorithm takes to reach an accuracy of  $10^{-3}$  as a function of the number of workers, where for each data point the algorithm parameters were re-optimized. For the SPARK implementations we start with one worker per machine and increase this number by one until we have used all available cores on our cluster. Due to significant memory overheads SPARK could not handle the data for less than 4 workers. For the MPI implementation we also scaled until we reached the maximum parallelism of our cluster. The dashed black line shows the theoretical performance of the MPI implementation where communication cost is assumed to be zero. We see that, when overheads are small, the COCoA algorithm can achieve a flat scaling behavior. However, present overheads related to communication have an increasing impact on the performance of COCoA as the number of workers increases. Hence, without minimizing overheads and carefully tuning the communication-computation trade-off, scaling of distributed optimization algorithms appears not to be feasible.

## 6. Conclusions

In this work we have studied the overheads of SPARK in the context of training general linear machine learning models. We have proposed several practical solutions for acceleration, and demonstrated a reduction of the performance gap between SPARK and MPI from  $20\times$  to less than  $2\times$ . While in this work we have chosen to focus on the COCoA algorithmic framework, our strategies for acceleration and overhead reduction can be applied to a variety of SPARK-based learning algorithms, including widely-use

mini-batch methods. We have demonstrated that the performance of the machine learning algorithms strongly depends on the choice of a parameter that controls the ratio of communication to computation. Furthermore, we have shown that the optimal choice of this parameter depends not only on the convergence properties of the underlying algorithm but also on the characteristics of the system on which it is executed. We conclude that in order to develop high-performance machine learning applications, one must carefully adapt the algorithm to account for the properties of the specific system on which such an application will be deployed. For this reason, algorithms that are able to automatically adapt their parameters to changes in system-level conditions are of considerable interest from a research perspective.

## 7. Acknowledgement

The authors would like to thank Frederick R. Reiss from the IBM Spark Technology Center / IBM Research - Almaden for highly constructive advice regarding this work and Martin Jaggi from EPFL for his help and inspiring discussions.

\* Java is a registered trademark of Oracle and/or its affiliates. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Other product or service names may be trademarks or service marks of IBM or other companies.

## References

- Armbrust, Michael, Das, Tathagata, Davidson, Aaron, Ghodsi, Ali, Or, Andrew, Rosen, Josh, Stoica, Ion, Wendell, Patrick, Xin, Reynold, and Zaharia, Matei. Scaling Spark in the Real World: Performance and Usability. *Proc. VLDB Endow.*, 8(12):1840–1843, August 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824080. URL <http://dx.doi.org/10.14778/2824032.2824080>.
- Bauschke, Heinz H and Combettes, Patrick L. *Convex Analysis and Monotone Operator Theory in Hilbert Spaces*. CMS Books in Mathematics. Springer New York, New York, NY, 2011.
- Dekel, Ofer, Gilad-Bachrach, Ran, Shamir, Ohad, and Xiao, Lin. Optimal Distributed Online Prediction Using Mini-batches. *J. Mach. Learn. Res.*, 13:165–202, January 2012. ISSN 1532-4435. URL <http://dl.acm.org/citation.cfm?id=2188385.2188391>.
- Dünner, Celestine, Forte, Simone, Takáč, Martin, and Jaggi, Martin. Primal-Dual Rates and Certificates. In *ICML 2016 - Proceedings of the 33th International Conference on Machine Learning*, 2016.
- Forum, Message P. MPI: A Message-Passing Interface Standard. *University of Tennessee*, 1994.
- Gittens, Alex, Devarakonda, Aditya, Racah, Evan, Ringenburgh, Michael F., Gerhardt, Lisa, Kottalam, Jey, Liu, Jialin, Maschhoff, Kristyn J., Canon, Shane, Chhugani, Jatin, Sharma, Pramod, Yang, Jiyan, Demmel, James, Harrell, Jim, Krishnamurthy, Venkat, Mahoney, Michael W., and Prabhat. Matrix Factorization at Scale: a Comparison of Scientific Data Analytics in Spark and C+MPI Using Three Case Studies. *CoRR*, abs/1607.01335, 2016. URL <http://arxiv.org/abs/1607.01335>.
- Hall, David, Ramage, Daniel, and et al. Breeze: A numerical processing library for Scala. <https://github.com/scalanlp/breeze>.
- Hunter, Tim. TensorFrames on Google’s TensorFlow and Apache Spark. *Bay Area Spark Meetup Salesforce*, Aug 2016.
- Jaggi, Martin, Smith, Virginia, Takáč, Martin, Terhorst, Jonathan, Krishnan, Sanjay, Hofmann, Thomas, and Jordan, Michael I. Communication-Efficient Distributed Dual Coordinate Ascent. In *NIPS 2014 - Advances in Neural Information Processing Systems 27*, pp. 3068–3076, 2014.
- Karau, Holden. Improving PySpark Performance: Spark performance beyond the JVM. *PyData, Amsterdam*, 2016.
- Li, Mu, Andersen, David G., Park, Jun Woo, Smola, Alexander J., Ahmed, Amr, Josifovski, Vanja, Long, James, Shekita, Eugene J., and Su, Bor-Yiing. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI’14, pp. 583–598, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL <http://dl.acm.org/citation.cfm?id=2685048.2685095>.
- Ma, Chenxin, Smith, Virginia, Jaggi, Martin, Jordan, Michael I, Richtárik, Peter, and Takáč, Martin. Adding vs. Averaging in Distributed Primal-Dual Optimization. In *ICML 2015 - Proceedings of the 32th International Conference on Machine Learning*, pp. 1973–1982, 2015.
- Meng, Xiangrui, Bradley, Joseph, Yavuz, Burak, Sparks, Evan, Venkataraman, Shivaram, Liu, Davies, Freeman, Jeremy, Tsai, D B, Amde, Manish, Owen, Sean, Xin, Doris, Xin, Reynold, Franklin, Michael J, Zadeh, Reza, Zaharia, Matei, and Talwalkar, Ameet. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.
- Microsoft. Multiverso. <https://github.com/Microsoft/multiverso>, 2015.
- Ousterhout, Kay, Rasti, Ryan, Ratnasamy, Sylvia, Shenker, Scott, and Chun, Byung-Gon. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 293–307, Oakland, CA, May 2015. USENIX Association. ISBN 978-1-931971-218. URL <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout>.
- Reyes-Ortiz, Jorge L., Oneto, Luca, and Davide, Anguita. Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. In *Procedia Computer Science*, NSDI’12, pp. 121–130. Elsevier, 2015.
- Richtárik, Peter and Takac, Martin. Distributed Coordinate Descent Method for Learning with Big Data. *arXiv*, October 2015.
- Shalev-Shwartz, Shai and Zhang, Tong. Stochastic Dual Coordinate Ascent Methods for Regularized Loss Minimization. *JMLR*, 14:567–599, February 2013.
- Smith, Virginia and Jaggi, Martin. PROXCOCO<sup>+</sup>. <https://github.com/gingsmith/proxcocoa>, 2015.
- Smith, Virginia, Forte, Simone, Jordan, Michael I, and Jaggi, Martin. L1-Regularized Distributed Optimization: A Communication-Efficient Primal-Dual Framework. *arXiv*, December 2015.
- Smith, Virginia, Forte, Simone, Ma, Chenxin, Takac, Martin, Jordan, Michael I, and Jaggi, Martin. CoCoA: A General Framework For Communication-Efficient Distributed Optimization. *arXiv*, November 2016.
- Walt, Sfan van der, Colbert, S. Chris, and Varoquaux, Gal. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2): 22–30, 2011. doi: <http://dx.doi.org/10.1109/MCSE.2011.37>. URL <http://scitation.aip.org/content/aip/journal/cise/13/2/10.1109/MCSE.2011.37>.
- Zaharia, Matei, Chowdhury, Mosharaf, Das, Tathagata, Dave, Ankur, Ma, Justin, McCauley, Murphy, Franklin, Michael J., Shenker, Scott, and Stoica, Ion. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI’12, pp. 2–2, Berkeley, CA, USA, 2012. USENIX Association. URL <http://dl.acm.org/citation.cfm?id=2228298.2228301>.
- Zinkevich, Martin A, Weimer, Markus, Smola, Alex J, and Li, Lihong. Parallelized Stochastic Gradient Descent. *NIPS 2010: Advances in Neural Information Processing Systems 23*, pp. 1–37, 2010.

## A. The CoCoA Algorithmic Framework

In this section we provide some background on the CoCoA Algorithmic Framework and its extensions CoCoA<sup>+</sup> (Ma et al., 2015) and PROXCOCO A<sup>+</sup> (Smith et al., 2015), which we refer to as CoCoA for simplicity, see (Smith et al., 2016) for a unified overview. This algorithm is designed to solve standard regularized loss minimization problems of the form

$$\min_{\alpha} f(A\alpha) + \sum_i r_i(\alpha_i). \quad (2)$$

in a distributed setting. Here  $f$  and  $r_i$  are convex functions,  $f$  is smooth,  $\alpha \in \mathbb{R}^n$  is the weight vector and  $A \in \mathbb{R}^{m \times n}$  is a given data matrix with column vectors  $\mathbf{c}_i \in \mathbb{R}^m$ ,  $i \in [n]$ .

The CoCoA algorithm is fundamentally based on the concept of Fenchel-Rockafellar duality which makes it more efficient than standard distributed frameworks. By forming a quadratic approximation to the global objective (2) CoCoA achieves separability of the problem over the coordinates of  $\alpha$  and the partitions. The resulting local subproblem have similar structure to the original problem and exploit second order information within the local data partition. Key to this framework is that the data local subproblems can be solved independently on each worker in parallel and only depend on the local data and the previously shared dual parameter vector  $\mathbf{w} := \nabla f(A\alpha) \in \mathbb{R}^m$ . As the data stays local during the whole algorithm and only a single  $m$ -dimensional vector is exchanged, the amount of communicated data is reduced to a minimum. CoCoA leaves it to the user to choose a local solver to find the solution of the local subproblems, which allows to reuse existing, fine-tuned solvers on the workers. Furthermore, CoCoA allows to solve the local subproblems to an arbitrary user-defined accuracy and hence allows to control the time spent in the local solver and trade-off communication and computation.

**Data Partitioning** We assume the dataset is distributed column-wise according to the partition  $\{\mathcal{P}_k\}_{k=1}^K$ . Hence columns  $\{\mathbf{c}_i\}_{i \in \mathcal{P}_k}$  of  $A$  reside on worker  $k$ . We denote the size of each partition by  $n_k = |\mathcal{P}_k|$ , the number of columns on this worker. Further, for  $k \in [K]$  we will use the notation  $\mathbf{v}_{[k]}$  to denote the vector with entries  $(\mathbf{v}_{[k]})_i = \mathbf{v}$  for  $i \in \mathcal{P}_k$  and  $(\mathbf{v}_{[k]})_i = 0$  for  $i \notin \mathcal{P}_k$ .

### A.1. Data-Local Subproblems

The data-local subproblem allocated to every machine  $k \in [K]$  have the following form:

$$\arg \min_{\Delta \alpha_{[k]}} \mathcal{G}_k^\sigma(\Delta \alpha_{[k]}; \mathbf{w}, \alpha_{[k]}) \quad (3)$$

where

$$\begin{aligned} \mathcal{G}_k^\sigma(\Delta \alpha_{[k]}; \mathbf{w}, \alpha_{[k]}) &:= \sum_{i \in \mathcal{P}_k} \lambda \left[ \frac{\eta}{2} (\alpha + \Delta \alpha_{[k]})_i^2 + (1 - \eta) |(\alpha + \Delta \alpha_{[k]})_i| \right] \\ &\quad + \frac{1}{K} \|\mathbf{w}\|_2^2 + \mathbf{w}^T A \Delta \alpha_{[k]} + \frac{\sigma}{2} \|A \Delta \alpha_{[k]}\|_2^2. \end{aligned} \quad (4)$$

Each machine  $k$  only works on its dedicated coordinates  $\alpha_{[k]}$  of  $\alpha$  and only needs access to data columns within the local partition  $\mathcal{P}_k$  as well as the previously shared dual vector  $\mathbf{w} := \nabla f(A\alpha)$ . The parameter  $\sigma$  of the subproblem characterizes the data partitioning and allows for more aggressive updates if correlations between partitions are small.

**Elastic Net** Ridge regression is a special case of the elastic net regularized problem

$$\min_{\alpha \in \mathbb{R}^d} \|A\alpha - \mathbf{b}\|_2^2 + \lambda \left[ \frac{\eta}{2} \|\alpha\|_2^2 + (1 - \eta) \|\alpha\|_1 \right] \quad (5)$$

with  $\eta \in [0, 1]$ ,  $\mathbf{b} \in \mathbb{R}^n$  are the labels and  $\lambda > 0$  the regularization parameter, we have the following local subproblems :

$$\begin{aligned} \mathcal{G}_k^\sigma(\Delta \alpha_{[k]}; \mathbf{w}, \alpha_{[k]}) &:= \sum_{i \in \mathcal{P}_k} \lambda \left[ \frac{\eta}{2} (\alpha + \Delta \alpha_{[k]})_i^2 + (1 - \eta) |(\alpha + \Delta \alpha_{[k]})_i| \right] \\ &\quad + \frac{1}{K} \|\mathbf{w}\|_2^2 + \mathbf{w}^T A \Delta \alpha_{[k]} + \frac{\sigma}{2} \|A \Delta \alpha_{[k]}\|_2^2. \end{aligned} \quad (6)$$

where  $\mathbf{w} := A\alpha - \mathbf{b} \in \mathbb{R}^n$ . For a problem independent definition of the local subproblems we refer to (Smith et al., 2016)

## A.2. Local Solver

We have chosen stochastic coordinate descent (SCD) as a local solver for our experiments. In every iteration updates are made by solving for a single coordinate exactly while keeping all others fixed. This algorithm is particularly nice in practice as the close-form updates are free of learning parameters. When solving the local subproblems in (6) using stochastic coordinate descent then coordinate update  $j$  can be computed as

$$\alpha_j^+ = \text{sign}(\tilde{\alpha}_j^+) [|\tilde{\alpha}_j^+| - \tau]_+ . \quad (7)$$

with

$$\tilde{\alpha}_j^+ = \frac{\sigma \|\mathbf{c}_j\|_2^2 \alpha_j - \mathbf{r}^\top \mathbf{c}_j}{\sigma \|\mathbf{c}_j\|_2^2 + \lambda n \eta} \quad (8)$$

where  $\mathbf{r}$  is the local residual. It is initialized in every round by the shared vector as  $\mathbf{r} = \mathbf{v}$  and updated as

$$\mathbf{r}^+ = \mathbf{r} + \sigma \mathbf{c}_j \Delta \alpha_j$$

after every coordinate step.

## A.3. Algorithm

Given this partitioning the detailed procedure of CoCoA for elastic net regularized regression using SCD as a local solver is given in Algorithm 1. Ridge regression is obtained when choosing  $\eta := 1$ . We refer to (Smith et al., 2015) for a description of the framework when using a general local solver. The core procedure is the following: At the beginning  $\alpha$  and the shared vector  $\mathbf{w}$  are initialized and every worker gets a local copy of  $\alpha$ . Then, in every round,  $\mathbf{w}$  is broadcast and the  $K$  workers independently work on their local subproblem. They update the local coordinates of the parameter vector  $\alpha$  and adapt  $\mathbf{w}$  accordingly. At the end of every round the workers communicate their change  $\Delta \mathbf{w}_k$  to the shared vector  $\mathbf{w}$  to the master node. The master aggregates these updates and determines the new  $\mathbf{w}$ .

---

### Algorithm 1 CoCoA for Ridge Regression using SCD as a local solver

---

- 1: **Data:** Data matrix  $A$  distributed column-wise according to partition  $\{\mathcal{P}_k\}_{k=1}^K$ .
  - Input:** Subproblem parameter  $\sigma$  and  $H$
  - Initialize:**  $\alpha^{(0)} := \mathbf{0} \in \mathbb{R}^n$ ,  $\mathbf{w}^{(0)} := -\mathbf{b} \in \mathbb{R}^d$
  - 2: **for**  $t = 0, 1, 2, \dots$  **do**
  - 3:   **for**  $k \in \{1, 2, \dots, K\}$  **in parallel over computers do**
  - 4:     run  $H$  steps of SCD on (6)  $\rightarrow \Delta \alpha_{[k]}$
  - 5:     update  $\alpha_{[k]}^{(t+1)} := \alpha_{[k]}^{(t)} + \Delta \alpha_{[k]}$
  - 6:     return  $\Delta \mathbf{w}_k := A \Delta \alpha_{[k]}$
  - 7:   **end for**
  - 8:   reduce  $\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} + \sum_{k=1}^K \Delta \mathbf{w}_k$
  - 9: **end for**
-